University of Setif 1- Ferhat Abbas

Faculy Of Sciences

Compter Science Department

UNIX SYSTEM ADMINISTRATION

1st Year Master Cyber Security

By Dr. Lyazid TOUMI

Contents

Reference Books

- UNIX and Linux System Administration Handbook (5th Ed.), Nemeth, Snyder, Hein, et al, Oreilly, 2017.
- The Practice of System and Network Administration (3rd Ed.), Limoncelli, Hogan, Chalup Addison-Wesly, 2017
- Essential System Administration (3rd Ed.), Æleen Frisch, Oreilly, 2022.
- Linux Bible (10th Ed.), Christopher Negus, Oreilly, 2020.
- The Linux Command Line, 2nd Ed. by William Shotts, Oreilly, 2019.

Chapter 1

Introduction to UNIX System Administration

Chapter Overview

This chapter provides a comprehensive introduction to the essential principles of UNIX system administration, equipping you with the foundational knowledge needed to manage and maintain UNIX-based systems effectively.

Key Focus Areas:

- The System Administrator's Role: Understand the responsibilities, challenges, and best practices for UNIX administrators, including system security, user management, and troubleshooting.
- Shell Proficiency: Discover why mastering the UNIX shell (Bash, Korn, or others) is critical for efficient administration, automation, and scripting.
- Core Administrative Tasks: Learn about vital operations such as process management, filesystem maintenance, backups, and network configuration.

Why UNIX Still Matters

Despite the rise of modern operating systems, UNIX remains a cornerstone of enterprise computing, powering servers, cloud infrastructure, and critical applications. We'll examine its enduring relevance, stability, and flexibility in today's IT landscape.

By the end of this chapter, you'll have a solid grasp of how system administrators keep UNIX environments secure, efficient, and reliable, ensuring seamless operation in professional settings.

1 The Role of a UNIX System Administrator

UNIX system administrators serve as the backbone of any organization running UNIX or UNIX-like systems. Their multifaceted responsibilities encompass all aspects of system operation, maintenance, and security.

1.1 Core Responsibilities

- System Installation and Configuration:
  - OS installation, patching, and upgrades
  - Kernel tuning and performance optimization
  - Filesystem hierarchy and storage management
  - Package and dependency management
- User and Security Management:
  - User account lifecycle management
  - Permission and access control (RBAC)
  - Authentication system configuration (PAM, LDAP)
  - Security auditing and compliance
- Network Services Administration:
  - TCP/IP stack configuration
  - Firewall (iptables/nftables) and routing management
  - Network service management (DNS, DHCP, NFS, Samba)
  - VPN and remote access configuration

1.2 The Administrator's Toolkit

A proficient UNIX administrator maintains expertise in several categories of essential tools:
Modern administrators also utilize configuration management tools (Ansible, Puppet, Chef) and containerization technologies (Docker, Kubernetes) in contemporary UNIX environments.

2 Importance of Shell Proficiency

While graphical tools exist, the command-line interface (CLI) remains the most powerful and ubiquitous administration method for UNIX systems, offering unparalleled control and flexibility.

Table 1: Essential UNIX Administration Tools

Tool Category	Representative Utilities
Process Management	ps, top, htop, kill, pkill, nice, renice, systemd
Filesystem Tools	df, du, mount, umount, fsck, lsblk, lsof, find
Network Utilities	ip, ss, netstat, ping, traceroute, dig, tcpdump, nmap
Security Tools	sudo, su, chmod, chown, chattr, iptables, auditd, fail2ban
System Monitoring	vmstat, iostat, sar, dmesg, journalctl, nagios, zabbix

2.1 Advantages of Shell Usage

1. Automation Capabilities:
   - Create reusable scripts for repetitive tasks
   - Implement complex workflows with conditionals and loops
   - Schedule automated maintenance jobs

Automated Backup Script

```
#!/bin/bash
# Automated backup script with logging and error handling
BACKUP_DIR="/backups"
LOG_FILE="/var/log/backup_$(date +\%Y\%m\%d).log"
SOURCE_DIR="/important_data"

{
    echo "Starting backup at $(date)"
    mkdir -p $BACKUP_DIR
    tar -czf "$BACKUP_DIR/$(date +\%Y\%m\%d).tar.gz"
    ↪  "$SOURCE_DIR" \
       && echo "Backup completed successfully" \
       || echo "Backup failed!"
    find $BACKUP_DIR -name '*.tar.gz' -mtime +30 -delete
    echo "Cleanup completed at $(date)"
} > $LOG_FILE 2>&1
```

2. Remote Administration:
   - Secure shell (SSH) for encrypted remote access

- rsync for efficient differential file transfers
- Terminal multiplexers (screen, tmux) for persistent sessions
- SSH key-based authentication for passwordless access

3. Scripting Power:
   - Combine utilities via pipes and redirection
   - Advanced text processing with awk, sed, and grep
   - Job scheduling with cron and systemd timers
   - Error handling and logging capabilities

2.2 Shell vs. GUI Administration

Table 2: Comparison of CLI and GUI Administration Methods

Feature	Command Line	GUI Tools
Resource Usage	Minimal CPU/RAM	Significant overhead
Remote Access	Full functionality via SSH	Often requires VNC/RDP
Automation	Complete scripting support	Limited to recorded macros
Reproducibility	Self-documenting scripts	Requires manual documentation
Precision	Exact control	Often abstracted operations
Speed	Immediate execution	Multiple clicks required

Key Insight: Professional system administrators typically use CLI for 90% of tasks, reserving GUI tools only for specific monitoring or configuration scenarios.

3 Key Administrative Tasks

3.1 System Monitoring and Performance Tuning

Effective administrators implement comprehensive monitoring strategies to maintain optimal system health:

- Performance Metrics Collection:

System Monitoring Commands

```
# CPU and process statistics
top -b -n 1 | head -n 20
mpstat -P ALL 1 5

# Memory utilization analysis
free -h
vmstat -SM 1 5

# Disk I/O monitoring
iostat -xmdz 1
iotop -o

# Network throughput
iftop -n -i eth0
nload eth0
```

- Log Management Framework:
  - Centralized logging with syslog-ng/rsyslog
  - Automated log rotation (logrotate)
  - Real-time monitoring with `tail -f` or `journalctl -f`
  - Log analysis tools (grep, awk, ELK Stack)

3.2 Security Practices and Hardening

Essential security measures for production systems:

1. Authentication and Access Control:
   - Password policies (/etc/login.defs, pam_pwquality)
   - SSH hardening (disable root login, change port)
   - Two-factor authentication implementation
   - Regular audit of sudo privileges (/etc/sudoers)
2. System Hardening Procedures:
   - Service minimization (disable unused services)
   - Filesystem permission audits (find / -perm /4000)
   - Regular security updates (yum-cron, unattended-upgrades)
   - Firewall configuration (iptables/nftables)
   - SELinux/AppArmor enforcement

3.3 Disaster Recovery Planning

A comprehensive disaster recovery strategy includes:

Table 3: Enterprise Backup Strategy Framework

Backup Type	Frequency	Implementation
Full System Image	Monthly	dd, Clonezilla
Incremental Files	Daily	rsync –link-dest
Configuration State	On Change	RCS, Git, etckeeper
Database Dumps	Hourly	mysqldump, pg_dump
Cloud Synchronization	Continuous	rclone, Duplicity

- Recovery Testing: Regular restore drills
- Offsite Storage: 3-2-1 backup rule (3 copies, 2 media, 1 offsite)
- Documentation: Detailed recovery procedures

Best Practice: Automate backup verification through checksum validation and periodic test restores.

Chapter Summary

This chapter introduced the critical role of UNIX system administrators and the tools they use. We emphasized the importance of shell proficiency and covered fundamental administrative tasks. The coming chapters will explore these concepts in greater depth, providing practical examples and advanced techniques.

Chapter 2

Understanding the UNIX Shell

Chapter Overview

This chapter offers a comprehensive exploration of the UNIX shell environment, a critical tool for system administrators and power users. We will examine:

Key Learning Objectives

- Shell Architecture: Understand the underlying structure and components of UNIX shells
- Shell Variants: Compare major shell types (Bourne, Bash, Zsh, Ksh) and their features
- Command Processing: Analyze the interpretation cycle from input to execution
- Shell Modes: Differentiate between interactive and non-interactive shell operations
- Environment Customization: Master configuration techniques for optimal productivity

Administrative Relevance

The chapter emphasizes practical shell skills essential for:

- Efficient system maintenance and troubleshooting
- Automation of repetitive administrative tasks
- Development of robust system management scripts
- Secure shell configuration for multi-user environments

Through practical examples and configuration case studies, you will gain the expertise needed to leverage the UNIX shell's full potential in professional system administration contexts.

1 Shell Architecture and Components

The UNIX shell operates as a sophisticated command interpreter with a modular architecture designed for flexibility and extensibility.

1.1 Core Components

- Command Parser:
  - Lexical analysis and tokenization
  - Special character handling (metacharacters, wildcards)
  - Syntax tree generation
- Variable Subsystem:
  - Environment variable management (`export`, `env`)
  - Shell variable expansion (`$VAR`, `${VAR}`)
  - Special parameters (`$0`, `$?`, `$$`)
- Process Controller:
  - Fork-exec mechanism
  - Job control (`fg`, `bg`, `jobs`)
  - Signal handling (`trap`, `kill`)
- I/O Redirection:
  - File descriptor management (`>`, `<`, `2>`)
  - Pipeline implementation (`|`)
  - Here-documents (`<<`)

1.2 Shell Startup Sequence

The shell initialization process follows a precise order when establishing the execution environment:

Shell Initialization Files

```
# System-wide configuration files (processed first)
/etc/profile        # Global environment and startup programs
/etc/bash.bashrc    # System-wide functions and aliases
/etc/environment    # Environment variable definitions

# User-specific files (processed in order of precedence)
~/.bash_profile     # Login shell initialization
~/.bash_login       # Alternative login configuration
~/.profile          # Fallback login configuration
~/.bashrc           # Non-login interactive shell configuration
~/.bash_logout      # Cleanup commands on shell exit
```

Note: The exact file processing order varies between shell types (Bash, Zsh, Ksh). Login shells process different files than non-login interactive shells.

2 Shell Types and Features

2.1 Major UNIX Shell Variants

Table 4: UNIX Shell Comparison and Characteristics

Shell	Key Features	Common Deployment
Bourne Shell (sh)	• Original UNIX shell (1977) • Basic scripting capabilities • Limited interactive features	• System startup scripts • POSIX-compliant environments
Bash (Bourne-Again SHell)	• sh-compatible with GNU extensions • Command history and editing • Arrays and associative arrays	• Default Linux shell • macOS terminal (until 10.15)
Korn Shell (ksh)	• Advanced scripting features • Floating-point arithmetic • Co-processes	• Enterprise UNIX systems • AIX default shell
C Shell (csh/tcsh)	• C-like syntax • Built-in arithmetic • Command history	• BSD systems • Scientific computing
Z Shell (zsh)	• Advanced completion system • Plugin and theme support • Shared command history	• Developer workstations • macOS default since 10.15

2.2 Shell Feature Matrix

Table 5: Detailed Shell Feature Comparison

Feature	sh	bash	ksh	csh	zsh
POSIX Compliance	Full	Partial	Full	No	Partial
Command History	✗	✓	✓	✓	✓
Job Control	✗	✓	✓	✓	✓
Associative Arrays	✗	v4.0+	✓	✗	✓
Floating-Point Math	✗	✗	✓	✓	✓
Plugin System	✗	✗	✗	✗	✓

Key Observations:

- Bash dominates Linux systems while ksh remains prevalent in enterprise UNIX
- Zsh offers the most interactive features but has higher resource usage
- For maximum portability, sh remains the safest choice for system scripts
- Modern shells (bash 5.0+, zsh) continue to add features like JSON support

3 Command Interpretation Process

The shell follows a sophisticated multi-stage process when interpreting and executing commands. This pipeline ensures proper handling of both simple commands and complex scripting constructs.

3.1 Processing Stages

1. Lexical Analysis:
   - Tokenization of input into words and operators
   - Identification of metacharacters and quoting
   - Comment removal
2. Expansion Phase (in order of execution):
   - Brace expansion: `{a,b,c}.txt → a.txt b.txt c.txt`
   - Tilde expansion: `~/user → /home/user`

- Parameter expansion: `$VAR`, `${VAR:-default}`
- Command substitution: `$(cmd)` or `` `cmd` ``
- Arithmetic expansion: `$((expression))`
- Process substitution: `<(cmd)`, `>(cmd)` (bash/zsh)
- Word splitting (using IFS)
- Filename expansion (globbing): `*.txt`

3. Redirection Handling:
   - File descriptor manipulation
   - Here-documents/here-strings
   - Pipeline creation
4. Execution Phase:
   - Built-in vs external command determination
   - PATH resolution
   - Process creation (fork-exec)

3.2 Expansion Examples

Advanced Shell Expansion Examples

```
#!/bin/bash
# Brace expansion
echo file-{1..3}.{txt,log}  # file-1.txt file-1.log file-2.txt...

# Nested command substitution
disk_usage=$(df -h $(mount | awk '/\/$/ {print $1}'))

# Advanced parameter expansion
default_path="/usr/local/bin"
echo "Using ${PATH:-$default_path}"

# Arithmetic in command substitution
timeout_seconds=$(( $(date +%s) + 3600 ))

# Process substitution
diff <(sort file1) <(sort file2)
```

Note: The expansion order is critical - brace expansion occurs before variable expansion, which occurs before word splitting. Quoting can suppress unwanted expansions.

4 Interactive vs. Non-Interactive Shells

UNIX shells operate in fundamentally different modes depending on their execution context, affecting both functionality and configuration.

4.1 Interactive Shell Characteristics

- User Interface:
  - Presents a command prompt (PS1 variable)
  - Supports line editing and command history
  - Provides tab completion and suggestions
- Behavior:
  - Reads commands from terminal device
  - Enables full job control (fg, bg, jobs)
  - Processes user-specific RC files (e.g., /.bashrc)
  - Handles signals (Ctrl-C, Ctrl-Z) interactively
- Use Cases:
  - Direct system administration
  - Development and debugging
  - Exploratory data analysis

4.2 Non-Interactive Shell Characteristics

- Execution Mode:
  - Reads commands from files or pipes
  - No visual prompt or line editing
  - Limited signal handling
- Configuration:
  - Processes only ENV file if specified
  - Skips most user-specific configurations
  - Often runs with restricted options (–norc, –noprofile)
- Use Cases:
  - Script execution
  - Automated batch processing
  - System startup/shutdown procedures

Table 6: Key Differences Between Shell Modes

Feature	Interactive Shell	Non-Interactive Shell
Command Source	Terminal input	Files/pipes
Prompt Display	Yes	No
RC File Processing	Full	Minimal
Job Control	Complete	Limited
Error Handling	Interactive	Exit on error
Default Options	-i flag	–norc

Administrative Note: Scripts should explicitly declare `#!/bin/bash -` to avoid inheriting interactive shell configurations that may cause unexpected behavior.

5 Shell Configuration and Customization

5.1 Environment Variables

Critical environment variables for professional system administration:

Essential Environment Variables

```
# System paths (order matters for security)
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
MANPATH=/usr/local/man:/usr/share/man

# Security-related settings
umask 027                          # Default file permissions
TMOUT=900                          # Session timeout (15 mins)
HISTCONTROL=ignoreboth             # Ignore duplicate commands
HISTIGNORE="&:ls:ll:la:cd:exit"    # Exclude trivial commands

# Application defaults
EDITOR=/usr/bin/vim                # Preferred text editor
PAGER=/usr/bin/less                # Paging program
PS1='[\u@\h \W]\$ '                # Custom prompt

# Development environment
LD_LIBRARY_PATH=/usr/local/lib     # Library paths
PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

5.2 Aliases and Functions

Professional-grade shell customizations:

Advanced Admin Customizations

```
# System monitoring aliases
alias meminfo='free -m -l -t'
alias cpuinfo='lscpu'
alias diskusage='df -h -x tmpfs -x devtmpfs'
alias openports='ss -tulnp'

# Safety nets
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Advanced functions
service_manage() {
    local service=$1
    local action=$2
    sudo systemctl $action $service
}

find_large_files() {
    find ${1:-.} -type f -size +${2:-10M} -exec ls -lh {} \+ | awk '{
     ↪  print \$9 ": " \$5 }'
}

# Git shortcuts (for systems with git)
if command -v git &>/dev/null; then
    alias gs='git status'
    alias gl='git log --oneline --graph --decorate'
fi
```

6 Shell Job Control

Comprehensive job management for administrators:

Table 7: Advanced Job Control Reference

Command	Function and Usage
`Ctrl+Z`	Suspend foreground job (SIGTSTP)
`bg [%job]`	Resume suspended job in background
`fg [%job]`	Bring job to foreground
`jobs -l`	List jobs with PID information
`kill -%n`	Send signal to job number n
`nohup command &`	Run command immune to hangups
`disown -h [%job]`	Remove job from shell's job table
`setsid command`	Run in new session (daemonize)

6.1 Job Control Best Practices

- Use `tmux` or `screen` for long-running processes
- Prefer `nohup` or `disown` for critical background jobs
- Monitor background jobs with `wait` in scripts
- Use process groups (`set -m`) for complex job control
- Consider `systemd-run` for persistent services

Note: Modern systems often use `systemd` for service management, but shell job control remains essential for interactive administration.

Chapter Summary

This chapter explored the UNIX shell's architecture, various shell types, and their features. We examined the command interpretation process and differences between interactive and non-interactive shells. The configuration techniques covered will help administrators customize their environment for maximum productivity. The next chapter will build on these concepts with shell scripting fundamentals.

Review Questions

1. Explain the three main phases of shell command interpretation
2. Compare and contrast Bash and Korn shell features
3. Describe how environment variables differ from shell variables

4. What are the key differences between interactive and non-interactive shells?

5. Create a .bashrc configuration with five useful aliases for system administration

Chapter 3

Shell Scripting Fundamentals

Chapter Overview

This chapter serves as a hands-on guide to mastering UNIX shell scripting, equipping system administrators with the skills to automate repetitive tasks, streamline system management, and enhance operational efficiency.

1 Script Basics and Structure

This section covers the fundamental building blocks of UNIX shell scripts, from basic components to interpreter selection. Understanding these core concepts is essential for writing effective and portable scripts.

1.1 Script Components

Every well-structured shell script contains these essential elements:

Basic Script Structure with Annotations

```
#!/bin/bash           # Shebang line (Section \ref{subsec:shebang})
# Script: backup.sh   # Metadata comments
# Author: Admin       # (Name, date, purpose)
# Description: Creates system backups

config="/etc/backup.conf"  # Variable declaration
LOGFILE="/var/log/backup.log"  # Constant convention

validate_input() {   # Function definition
    [ -f "$config" ] || {
        echo "Error: Config missing" >&2
        return 1
    }
}
main() {             # Main program logic
    validate_input || exit 1
    tar -czf "/backup/$(date +\%F).tar.gz" /target
}
main "$@"           # Execution entry point
exit 0              # Explicit exit status (Section
 ↪ \ref{subsec:exit-status})
```

Key components explained:

- Shebang Line: Mandatory first line specifying the interpreter (see Section 1.2)
- Metadata: Comments documenting purpose, author, and version
- Variable Scope:
  - `UPPERCASE` for constants
  - `lowercase` for local variables
- Modular Design: Functions for reusable code blocks
- Error Handling: Explicit status checks and exits

1.2 Shebang Variations

The shebang (`#!`) determines script execution behavior. Choose carefully based on:

Critical considerations:

- Portability: `env` finds the interpreter in user's PATH
- Security: Hard paths prevent interpreter hijacking

Table 8: Common Shebang Interpreters and Use Cases

Type	Advantages	**Example**
Absolute Path	Guaranteed interpreter location	`#!/bin/bash`
env Lookup	Portable across systems	`#!/usr/bin/env bash`
POSIX Mode	Cross-shell compatibility	`#!/bin/sh`
Language Specific	Direct execution without wrapper	`#!/usr/bin/perl`

- Features: Bashisms (`[[ ]]` arrays) won't work in `/bin/sh`

1.3 Exit Status Conventions

Standard exit codes and their meanings:

Table 9: Reserved Exit Status Codes

Code	Meaning
0	Success
1	General error
2	Misuse of shell builtins
126	Command cannot execute
127	Command not found
128+N	Terminated by signal N

Best practices:

- Always return explicit statuses: `return 0` or `exit 1`
- Document custom codes (64-113 available for user-defined errors)
- Use constants: `readonly E_CONFIG=78`

2 Variables and Data Types

This section covers shell variable handling, data typing conventions, and special built-in variables essential for script development.

2.1 Variable Handling

Shell variables follow specific declaration and scoping rules:

Variable Assignment

```
# Basic assignment (no spaces around =)
username="admin"!\label{line:basic_assign}!
```

Key concepts:

- Assignment: No spaces around `=`
- Constants: Use `readonly` or `declare -r`
- Scoping: `local` limits visibility to functions
- Typing: `declare` options:
  - `-i` for integers
  - `-a` for arrays
  - `-r` for read-only

2.2 Special Variables

Shell provides automatic variables for script control:

Table 10: Predefined Shell Variables with Examples

Variable	Purpose	**Example Usage**
`$0`	Current script name	`echo "Running: $0"`
`$1--$9`	Positional parameters	`backup_file=$1`
`$\*`	All arguments as single string	`log_args.sh "$\*"`
`$@`	All arguments as separate strings	`process_files "$@"`
`$#`	Argument count	`[ $# -eq 0 ] && usage`
`$?`	Last command exit status	`mkdir tmp || echo "Failed: $?"`
`$$`	Current process PID	`LOG="$$.log"`
`$!`	Last background process PID	`kill $!`

Critical notes:

- Parameter Expansion: Always quote variables: `"$@"` vs `$*`
- Default Values: Use `:-` syntax: `$VAR:-default`
- Error Handling: Check `$?` immediately after commands
- Process Control: `$$` for temp files, `$!` for job control

2.3 Data Type Considerations

Table 11: Shell Variable Typing Behaviors

Type	Declaration	Behavior
String	Default (untyped)	Text processing, expansion
Integer	`declare -i`	Arithmetic without `expr`
Array	`declare -a`	Indexed lists (0-based)
Associative Array	`declare -A`	Key-value pairs (Bash 4+)
Read-only	`readonly` or `declare -r`	Immutable after assignment

Practical examples:

Typed Variable Usage

```
declare -i total=0
total+=5  # No $ required for arithmetic

declare -A services=(
    [web]="nginx"
    [db]="postgresql"
)
echo "${services[web]}"  # Outputs: nginx
```

3 Control Structures

This section covers the fundamental control flow mechanisms in shell scripting, enabling decision-making and repetitive operations.

3.1 Conditional Statements

Shell scripts support several conditional execution patterns:

Conditional Statements with Best Practices

```
#!/bin/bash

# Basic if-elif-else (POSIX compliant)
if [ -f "/etc/passwd" ]; then!\label{line:if-file}!
    echo "System user file exists"
elif [ -d "/etc" ]; then!\label{line:elif-dir}!
    echo "/etc directory exists but passwd missing"
else
    echo "Critical system files missing" >&2!\label{line:stderr}!
    exit 1
fi!\label{line:fi}!

# Modern Bash conditional ([[ ]] with regex)
if [[ "$OS" =~ ^[Ll]inux ]]; then!\label{line:regex}!
    echo "Linux variant detected"
fi

# Case statement (pattern matching)
case $1 in!\label{line:case}!
    start--start)
        service start
        ;;
    stop--stop)
        service_stop
        ;;
    status|--status)
        service_status
        ;;
    *)!\label{line:case-default}!
        echo "Usage: $0 {startstopstatus}" >&2
        exit 2
        ;;
esac!\label{line:esac}!
```

Key features:

- Test Operators:
  - `[ ]`: POSIX-compliant (spaces required)
  - `[[ ]]`: Bash extension (safer, supports regex)
- Error Handling:
  - Redirect errors to `stderr` (Line ??)
  - Use meaningful exit codes (Line ??)

- Pattern Matching:
  - `case` for multiple patterns
  - Regex support in `[[ ]]`

3.2 Loop Constructs

Shell provides three primary looping mechanisms:

Loop Structures with Practical Applications

```
#!/bin/bash

# For loop (iterating lists)
for package in nginx postgresql redis; do!\label{line:for-list}!
    if ! which "$package" &>/dev/null; then
        apt-get install -y "$package"
    fi
done

# While loop (stream processing)
while IFS= read -r line; do!\label{line:while-read}!
    [[ "$line" =~ ^# ]] && continue  # Skip comments
    process_log_entry "$line"
done < /var/log/app.log!\label{line:done-redirect}!

# Until loop (polling)
attempt=0
until [ $attempt -ge 3 ] || db_connection_test;
 ↪  do!\label{line:until}!
    sleep $((attempt * 2))
    ((attempt++))
done!\label{line:until-done}!

# C-style for loop
for ((i=0; i<10; i++)); do!\label{line:c-style}!
    echo "Iteration $i"
done
```

Loop control techniques:

- List Processing:
  - Word splitting in `for` (Line ??)
  - Safe reading with `IFS=` and `-r` (Line ??)

- Flow Control:
  - `continue` to skip iterations
  - `break` to exit loops early
- Redirection:
  - File input redirection
  - Process substitution `< <(cmd)`

3.3 Best Practices

Table 12: Control Structure Anti-Patterns vs Recommended Approaches

Avoid	Prefer
`if [ $var ]` (empty string check)	`if [ -n "$var" ]` (explicit test)
`for f in \`ls\`` (unquoted command sub)	`for f in *` (globbing) or `while read`
Nested `if` chains	`case` statements or early returns
Infinite `while true` loops	Timeout mechanisms: `timeout` command

Advanced patterns:

Parallel Processing Example

```
# Process files in parallel (Bash 4+)
max_jobs=4
for file in *.log; do
    ((current_jobs >= max_jobs)) && wait -n
    process_file "$file" &
    ((current_jobs++))
done
wait  # Cleanup remaining jobs
```

4 Functions and Modularity

This section covers shell function implementation strategies for creating maintainable, reusable scripts. Proper function design is crucial for script organization and debugging.

4.1 Function Definition and Usage

Advanced Function Techniques

```
#!/bin/bash

# Self-documenting function (metadata in comments)
# @desc: Checks disk space with optional threshold
# @usage: check_disk_usage [threshold_percent]
# @param: $1 - Warning threshold (default: 90)
check_disk_usage() {
    local threshold=${1:-90}!\label{line:local-default}!
    local usage=$(df -h --output=pcent,target | tail -n +2)

    while read -r pct mount; do!\label{line:while-read}!
        pct=${pct%\%}
        (( pct >= threshold )) && {
            echo "WARNING: $mount at ${pct}%" >&2
            return 1
        }
    done <<< "$usage"!\label{line:here-string}!

    return 0
}

# Library function (sourced from external file)
# @desc: Validates email format
# @usage: validate_email "address"
validate_email() {
    [[ "$1" =~ ^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$ ]]
}

# Main execution flow
if check_disk_usage 95; then!\label{line:func-call}!
    echo "Disk space OK"
else
    exit 1
fi
```

Key features:

- Parameter Handling:
  - Default values
  - Named parameters via `$1`, `$2`, etc.

- Data Processing:
  - Here strings for input
  - Process substitution alternatives
- Return Values:
  - Explicit status codes (0=success)
  - Boolean patterns

4.2 Function Best Practices

4.2.1 Implementation Guidelines

- Variable Scope:
  - Always declare `local` variables
  - Avoid global state modifications
- Error Handling:
  - Use `set -euo pipefail` in functions
  - Implement cleanup traps
- Documentation:
  - Include usage examples
  - Document parameters and return values

4.2.2 Design Patterns

Table 13: Common Shell Function Patterns

Pattern	Implementation
Validation	Return 0/1 with error messages to stderr
Logger	Centralized logging function with levels
Wrapper	Function that modifies command behavior
Callback	Pass function names as parameters

Example of advanced patterns:

Logger Implementation

```
#!/bin/bash

# Logging library function
log() {
    local level=$1
    local message=$2
    local timestamp=$(date +"%Y-%m-%d %T")

    case $level in
        INFO) color="\033[0;32m" ;;
        WARN) color="\033[0;33m" ;;
        ERROR) color="\033[0;31m" ;;
    esac

    echo -e "${color}${timestamp} [${level}] ${message}\033[0m" >&2
}

# Usage
log INFO "Processing started"
log ERROR "Invalid configuration" && exit 1
```

4.3 Modular Script Design

- Source Organization:
  - Separate functions into `lib/*.sh` files
  - Main script under `bin/`
- Namespace Management:
  - Prefix related functions (`file_`, `db_`)
  - Use `source` for library inclusion
- Dependency Control:
  - Verify function availability
  - Implement version checks

Example project structure:

```
/usr/local/bin/myapp
/usr/local/lib/myapp/
├── utils.sh    # Common functions
├── config.sh   # Configuration loader
└── modules/    # Feature-specific functions
```

5 Input/Output Handling

This section covers advanced techniques for managing script input/output streams, including redirection, piping, and document embedding.

5.1 Redirection Operators

Table 14: Advanced I/O Redirection Reference

Operator	Effect	**Example**
`> file`	Overwrite file with stdout	`ls > dir_contents.txt`
`>> file`	Append stdout to file	`echo $result >> log.txt`
`2> file`	Redirect stderr only	`rm badfile 2> errors.log`
`2>&1`	Combine stderr with stdout	`cmd > log 2>&1`
`&> file`	Redirect both streams (Bash)	`./script &> output.log`
`< file`	Use file as stdin	`sort < data.txt`
`<<<`	Here string (Bash)	`grep "text" <<< $var`
`|`	Pipe between commands	`ps aux | grep ssh`
`>| file`	Force overwrite (noclobber)	`echo "data" >| file`
`<> file`	Read/write same file	`exec 3<> lockfile`

Critical nuances:

- Stream Numbers: 0=stdin, 1=stdout, 2=stderr
- Order Matters: `2>&1 >file` differs from `>file 2>&1`
- File Descriptors: Create custom (3-9) for complex I/O

5.2 Here Documents

Advanced Here Document Techniques

```
#!/bin/bash

# Basic heredoc with variable expansion
cat <<CONFIG > app_settings.conf
# Generated $(date)
DB_HOST=${DB_HOST:-localhost}
DB_PORT=${DB_PORT:-5432}
MAX_RETRIES=3
CONFIG

# Indented heredoc (Bash 4+)
cat <<-EOF | while read -r line; do
    \tThis line preserves tabs
    \tBut leading tabs are stripped
EOF
    echo "Processing: $line"
done

# Literal heredoc (no expansion)
ssh server <<'REMOTE_CMDS'  # Single quotes matter
sudo apt update
sudo apt upgrade -y
REMOTE_CMDS

# Heredoc to variable
read -r -d '' help_msg <<HELP
Usage: $0 [options]
  -h  Show this help
  -v  Enable verbose mode
HELP
```

Key features:

- Delimiter Choice: Uppercase markers (EOF, CONFIG) for visibility
- Expansion Control:
  - Quoted delimiters disable expansion
  - Unquoted allows variables/commands
- Indentation: `<<-` strips leading tabs (not spaces)

5.3 File Descriptor Management

Advanced FD Manipulation

```
# Multiple output streams
{
    echo "Header"
    ls /nonexistent 2>&3
} 1>output.log 3>errors.log

# Persistent file descriptors
exec 4<database.sql  # Open for reading
while read -u4 line; do
    parse_sql "$line"
done
exec 4<&-  # Close descriptor

# Network redirection
exec 5<>/dev/tcp/google.com/80
echo -e "GET / HTTP/1.1\nHost: google.com\n\n" >&5
cat <&5
```

Best practices:

- Cleanup: Always close custom FDs with `<&-` or `>&-`
- Atomic Writes: Use `> tmpfile && mv tmpfile target` for safety
- Buffering: `stdbuf` utility controls stream buffering

5.4 Process Substitution

Comparing Files Without Temp Files

```
# Compare sorted outputs
diff <(sort file1) <(sort file2)

# Multi-input processing
paste <(cut -f1 data.tsv) <(cut -f3 data.tsv)

# Output capture
while read -r result; do
    process "$result"
done < <(complex_cmd --options)
```

Advantages:

- Avoids temporary files
- Enables parallel processing
- Maintains variable scope

6 Error Handling and Debugging

This section covers professional techniques for building resilient shell scripts and effective debugging methodologies.

6.1 Error Prevention Techniques

Enterprise-Grade Script Hardening

```
#!/bin/bash
# Strict execution mode (recommended set)
set -euo pipefail!\label{line:strict-mode}!
shopt -s failglob # Fail on unmatched globs
# Comprehensive dependency check
require_commands() {!\label{line:require-cmds}!
    local missing=()
    for cmd in "$@"; do
        if ! command -v "$cmd" &>/dev/null; then
            missing+=("$cmd")
        fi
    done
    ((${#missing[@]})) && {!\label{line:missing-check}!
        echo "Missing dependencies:" >&2
        printf ' - %s\n' "${missing[@]}" >&2
        exit 1
    }
}

# Validated argument processing
parse_arguments() {!\label{line:arg-parser}!
    [[ "$#" -ge 2 ]]  {
        echo "Usage: ${0##*/} <input> <output>" >&2
        exit 2
    }
    [[ -r "$1" ]]  {!\label{line:file-check}!
        echo "Cannot read input file: $1" >&2
        exit 3
    }
    # Safe output redirection check
    if ! touch "$2" &>/dev/null; then!\label{line:write-check}!
        echo "Cannot write to output: $2" >&2
        exit 4
    fi
}
# Main execution with error trapping
main() {
    require_commands rsync jq date!\label{line:main-requires}!
    parse_arguments "$@"
    # Business logic here
    process_data "$1" "$2"
}
main "$@"
```

Key hardening practices:

- Strict Mode (Line ??):
  - `-e`: Exit on error
  - `-u`: Fail on unset variables
  - `-o pipefail`: Catch pipeline errors
- Dependency Management (Lines ??-??):
  - Early validation of required tools
  - User-friendly error reporting
- Input Validation (Lines ??-??):
  - Argument count checking
  - File accessibility verification

6.2 Debugging Methods

6.2.1 Systematic Debugging Approaches

Table 15: Debugging Techniques Matrix

Method	Implementation	Use Case
Trace Mode	`set -x` or `bash -x script.sh`	Step-by-step execution
Error Traps	`trap 'echo Error in $0 at $LINENO' ERR`	Contextual error logging
Syntax Check	`bash -n script.sh`	Pre-execution validation
Verbose Mode	`set -v`	Show raw input processing
Custom Logging	`exec 3> debug.log; PS4='+ $(date) '`	Timestamped tracing

6.2.2 Advanced Tracing Example

Production-Grade Debugging

```
#!/bin/bash

# Configure debug output
DEBUG_LOG="${TMPDIR:-/tmp}/script_debug.$$"
exec 5> "$DEBUG_LOG"!\label{line:debug-fd}!
BASH_XTRACEFD="5"!\label{line:xtracefd}!

# Enhanced trace formatting
PS4='+[${LINENO}][${FUNCNAME[0]:-main}]: '!\label{line:ps4}!
set -x

# Debug trap with backtrace
trap 'echo "Error at ${LINENO}: ${BASH_COMMAND}" >&5;\
      echo "Call stack:\n${FUNCNAME[@]}" >&5'
       ↪  ERR!\label{line:err-trap}!

# Function demonstrating tracing
process_item() {
    local item=$1
    [[ -z "$item" ]] && return 1

    echo "Processing: $item"
}

main() {
    process_item "$1"
}

main "$@"
```

Debugging best practices:

- Isolated Logging (Lines ??-??):
  - Separate debug output from application streams
  - Use unique filenames (`$$` for PID)
- Enhanced Tracing (Line ??):
  - Include line numbers and function names
  - Add timestamps for performance analysis
- Error Context (Line ??):

- Capture failing command and location
- Log call stack for complex scripts

6.3 Error Recovery Patterns

Resilient Execution Framework

```bash
#!/bin/bash
# Global error handler
trap 'cleanup ${LINENO} "${BASH_COMMAND}"' EXIT ERR
cleanup() {
    local line=$1 cmd=$2
    local status=$?
    (( status == 0 )) && return
    echo "CRITICAL: Failure at line $line: '$cmd'" >&2
    echo "Exit status: $status" >&2
    # Resource cleanup
    rm -f "$TEMP_FILE"
    [[ -n "$DB_CONN" ]] && disconnect_db
    exit $status
}
# Temporary file handling
TEMP_FILE=$(mktemp) || {
    echo "Temp file creation failed" >&2
    exit 1
}
# Business logic with automatic cleanup
process_data() {
    # Operations that may fail
    transform_data > "$TEMP_FILE"
    load_to_database "$TEMP_FILE"
}
main() {
    initialize_db_connection
    process_data
}
main "$@"
```

Recovery strategies:

- Resource Guarantees:
  - `trap` on both `EXIT` and `ERR`
  - Atomic file operations

- State Management:
  - Clean up temporary resources
  - Close network connections
- Diagnostic Data:
  - Preserve error context
  - Log environment state

7 Advanced Scripting Techniques

This section explores powerful shell programming features for complex data processing and system automation tasks.

7.1 Arrays and Associative Arrays

7.1.1 Indexed Arrays

Professional Array Patterns

```bash
#!/bin/bash

# Safe array initialization
log_files=()
while IFS= read -r -d $'\0' file; do
    log_files+=("$file")  # Append null-delimited finds
done < <(find /var/log -name '*.log' -print0)

# Array operations
echo "Found ${#log_files[@]} log files"  # Count elements
printf ' - %s\n' "${log_files[@]}"      # Iterate safely
sorted=($(printf '%s\n' "${log_files[@]}" | sort -r)) # Sort

# Slice and subset patterns
recent_files=("${log_files[@]:0:5}")  # First 5 elements
error_files=(${log_files[@]/*error*})  # Subset matching

# Parallel processing
for file in "${recent_files[@]}"; do
    analyze_log "$file" &
done
wait
```

Key Features:

- Null-Delimited Safety: Robust handling of filenames with spaces
- Memory Efficiency: Streaming into arrays without subshells
- Modern Substitution: Pattern-based filtering and slicing

7.1.2 Associative Arrays (Bash 4+)

Configuration Management Example

```
declare -A service_config=(
    [port]="8080"
    [timeout]="300"
    [max_connections]="1000"
)

# Dynamic configuration building
read_config() {
    while IFS='=' read -r key value; do
        [[ $key ]] && service_config["$key"]="$value"
    done < config.properties
}

# Validation check
validate_config() {
    required_keys=(port timeout)
    for key in "${required_keys[@]}"; do
        [[ -v service_config[$key] ]] || {
            echo "Missing $key in config" >&2
            return 1
        }
    done
}
```

Common Use Cases:

- Configuration management systems
- Key-value data processing
- State tracking in complex scripts

7.2 Process Substitution

Advanced Data Pipelines

```
#!/bin/bash

# Real-time log analysis
analyze_access() {
    awk '{print $1}' <(tail -F /var/log/nginx/access.log) | \
    sort | uniq -c | sort -nr
}

# Database ETL pattern
extract_transform() {
    psql -c "COPY (SELECT * FROM sales) TO STDOUT CSV" | \
    awk -F, '{if($3 > 1000) print $1,$2*1.08}' > \
    >(gzip > processed_$(date +\%F).csv.gz)
}

# Multi-stream processing
compare_servers() {
    paste \
        <(ssh web1 "vmstat 1 5 | tail -4") \
        <(ssh web2 "vmstat 1 5 | tail -4") | \
    column -t
}

# Secure temporary processing
encrypt_data() {
    gpg --encrypt \
        <(tar cz sensitive_data) \
        > archive_$(date +\%s).tar.gz.gpg
}
```

Performance Considerations:

- Memory Efficiency: Avoids intermediate files
- Parallelism: Enables concurrent processing
- Stream Security: Encrypted pipeline patterns

7.3 Named Pipes and Coprocesses

Inter-Process Communication

```
# Named pipe for persistent IPC
PIPE=/tmp/mypipe
mkfifo "$PIPE"

# Writer process
while true; do
    sensors > "$PIPE"
    sleep 5
done &

# Reader process
while read -r data; do
    check_temperatures "$data"
done < "$PIPE"

# Coprocess for stateful interaction
coproc DB_CONN {
    psql -U user -d inventory
}

# Send query
echo "SELECT * FROM products;" >&${DB_CONN[1]}

# Read results
read -u ${DB_CONN[0]} -r result
```

8 Script Security Considerations

This section covers essential security practices for production-grade shell scripting, addressing common vulnerabilities and hardening techniques.

Table 16: Security Configuration Checklist

Category	Insecure Pattern	Secure Alternative
File Handling	`rm -rf $TMP/*`	`rm -rf -- "$TMP"/*`
Command Execution	`eval $user_input`	`case $user_input in safe_cmd) ... ;; esac`
Temporary Files	`> /tmp/data`	`mktemp -p /secure/tmp`
Privileges	`sudo cmd`	`sudo --non-interactive -- cmd`

8.0.1 System Hardening

8.1 Example Secure Script

Hardened Database Backup Script

```
#!/bin/bash
set -euo pipefail
shopt -s failglob  # Fail on failed glob expansion
# Environment hardening
export PATH="/bin:/usr/bin:/sbin:/usr/sbin"
umask 077  # Restrict new file permissions
# Validate credentials file
readonly CRED_FILE="${HOME}/.dbcreds"
[[ -f "$CRED_FILE" ]] || { echo "Credentials missing" >&2; exit 1; }
[[ "$(stat -c %a "$CRED_FILE")" == 600 ]] || {
    echo "Insecure credentials permissions" >&2
    exit 1}
# Validate and sanitize input
validate_hostname() {
    [[ "$1" =~ ^[a-zA-Z0-9.-]+$ ]] || return 1
    getent hosts "$1" >/dev/null || return 1}
if (( $# != 1 )) || ! validate_hostname "$1"; then
    echo "Usage: $0 <valid_hostname>" >&2
    exit 2
fi
# Secure temporary directory
readonly TMP_DIR=$(mktemp -d -p /secure/tmp backup.XXXXXXXXXX)
trap 'rm -rf -- "$TMP_DIR"' EXIT INT TERM
# Database operations with credential protection
read_db_password() {
    local pass
    IFS= read -r pass < <(
        gpg --quiet --decrypt "$CRED_FILE" 2>/dev/null
    ) || {
        echo "Credential decryption failed" >&2
        exit 3}
    printf '%s' "$pass"}
# Execute with minimal privileges
if [[ $EUID -eq 0 ]]; then
    exec setpriv --reuid=backupuser --init-groups "$0" "$@"
fi
# Main backup operation
PGPASSWORD=$(read_db_password) pg_dump -h "$1" -U backupuser \
    | gzip -9 > "$TMP_DIR/dump.sql.gz" || {
    echo "Backup failed" >&2
    exit 4}
```

Security Features Implemented:

- Privilege Separation:
  - Automatic privilege dropping for operations
  - Dedicated system user context
- Credential Protection:
  - Encrypted credential storage
  - Secure password handling without persistence
- Secure Execution:
  - Restricted PATH environment
  - Atomic temporary file handling
  - Network security enforcement

8.2 Common Vulnerabilities and Mitigations

Table 17: Shell Script Security Risks

Vulnerability	Example	Solution
Shell Injection	`rm $user_input`	`rm -- "$user_input"`
Race Conditions	`> /tmp/file`	`mktemp + O_EXCL`
TOCTOU Issues	`[ -f $f ] && rm $f`	Atomic operations
Information Leak	`ps aux | grep pass`	Credential sanitization

8.3 Advanced Security Patterns

Audit Logging Framework

```
#!/bin/bash
set -o functrace  # Trace function calls
shopt -s extdebug  # Enable debugging hooks

# Security audit log configuration
readonly AUDIT_LOG="/var/log/script_audit.log"
exec 5>> "$AUDIT_LOG"  # Dedicated file descriptor

# Log security events
log_audit_event() {
    printf '[%s] [UID=%d] %s\n' \
        "$(date --utc +%Y-%m-%dT%H:%M:%SZ)" \
        "$EUID" \
        "$*" >&5
}

# Command execution wrapper
secure_exec() {
    log_audit_event "Executing: $*"
    command "$@" || {
        log_audit_event "Failed (status=$?): $*"
        return 1
    }
}

# Intercept dangerous commands
alias rm='secure_exec rm --preserve-root --one-file-system'
```

Chapter Summary

This chapter covered essential shell scripting techniques from basic syntax to advanced features. We examined variables, control structures, functions, and robust error handling. The security practices and debugging methods will help create maintainable administration scripts.

Practical Exercises

1. Create a script that checks disk space and emails alerts when below threshold
2. Write a user management script with add/delete/list functions
3. Develop a log rotation script with compression and retention policy
4. Implement a network port scanner using shell functions
5. Build a configuration file generator using here documents

Administration Exercises

1. Create a script that monitors disk space and emails alerts when usage exceeds 90%
2. Write a command pipeline to identify the top 5 memory-consuming processes
3. Configure user accounts with appropriate permissions for a shared development environment
4. Analyze web server logs to identify the most frequent visitors
5. Troubleshoot a network connectivity issue using the commands covered

Chapter 4

Advanced Shell Features

Chapter Overview

This chapter explores sophisticated shell capabilities that enable administrators to build powerful, efficient solutions. We'll cover I/O redirection, process substitution, coprocesses, advanced parameter expansion, and shell customization techniques for professional-grade scripting.

1 Advanced I/O Redirection

1.1 File Descriptor Manipulation

Table 18: File Descriptor Reference

FD	Name	Description
0	stdin	Standard input
1	stdout	Standard output
2	stderr	Standard error
3-9		Additional descriptors

FD Operations

```
# Redirect stderr to stdout
command 2>&1

# Redirect both to file
command &> output.log

# Custom file descriptors
exec 3<> /tmp/lockfile  # Open FD 3 for RW
echo "locked" >&3
exec 3>&-               # Close FD 3

# Redirect to multiple destinations
{ echo "Message"; ls /nonexistent; } 2>&1 | tee output.log
```

1.2 Here Documents and Strings

Here Document Techniques

```
# Indented here document (Bash 4+)
cat <<-EOF
    This text will have
    leading tabs removed
EOF

# Parameter expansion in here string
tr 'a-z' 'A-Z' <<< "$USER"

# Execute remote commands via SSH
ssh server.example.com <<'EOSSH'
    sudo apt update
    sudo apt upgrade -y
EOSSH
```

2 Process Substitution and Coprocesses

2.1 Process Substitution

Process Substitution Examples

```
# Compare sorted outputs
diff <(sort file1) <(sort file2)

# Multi-input processing
join <(cut -f1 data1) <(cut -f2 data2)

# Avoid temporary files
paste -d: <(cut -d: -f1 /etc/passwd) \
           <(cut -d: -f3 /etc/passwd)
```

2.2 Coprocesses

Coprocess Communication

```
# Start coprocess
coproc DB_CONN {
    mysql -u admin -p"$DB_PASS" inventory
}

# Send query
echo "SELECT * FROM products;" >&${DB_CONN[1]}

# Read results
mapfile -t results <&${DB_CONN[0]}
echo "${results[@]}"
```

3 Advanced Parameter Expansion

3.1 Pattern Matching Operators

Table 19: Parameter Expansion Operators

Expression	Effect
`${var%pattern}`	Remove shortest suffix match
`${var%%pattern}`	Remove longest suffix match
`${var#pattern}`	Remove shortest prefix match
`${var##pattern}`	Remove longest prefix match
`${var//pattern/repl}`	Global replacement
`${var:offset:length}`	Substring expansion
`${var:-default}`	Use default if unset

Parameter Expansion Examples

```
# Convert path to filename
filename=${fullpath##*/}

# Change file extension
newfile=${file%.*}.bak

# Default values
backup_dir=${BACKUP_DIR:-/var/backups}

# Case conversion
upper=${filename^^}
lower=${filename,,}

# Array slicing
subarray=("${files[@]:2:5}")
```

4 Arrays and Associative Arrays

4.1 Advanced Array Techniques

Array Operations

```
# Indexed array
services=("nginx" "mysql" "redis")

# Associative array (Bash 4+)
declare -A ports=(
    [http]=80
    [https]=443
    [ssh]=22
)

# Multi-dimensional simulation
declare -A servers
servers[web,primary]="web1.example.com"
servers[web,backup]="web2.example.com"

# Array manipulation
all_services=("${services[@]}" "${!ports[@]}")
```

4.2 Array Processing

Array Processing Examples

```
# Read files into array
mapfile -t lines < config.cfg

# Filter array
readarray -t log_files < <(find /var/log -type f -name "*.log")

# Join array elements
printf -v joined '%s,' "${services[@]}"
echo "${joined%,}"  # Remove trailing comma

# Array intersection
comm -12 <(printf '%s\n' "${array1[@]}" | sort) \
         <(printf '%s\n' "${array2[@]}" | sort)
```

5 Shell Options and Customization

5.1 set and shopt Commands

Table 20: Useful Shell Options

Option	Effect
`set -o nounset`	Treat unset variables as errors
`set -o pipefail`	Pipeline exit status is last failure
`shopt -s globstar`	Enable ** recursive globbing
`shopt -s dotglob`	Include hidden files in globs
`shopt -s extglob`	Enable extended pattern matching
`shopt -s nocaseglob`	Case-insensitive globbing

5.2 Custom Prompt Engineering

Advanced Prompt Customization

```
# Git-aware prompt
PS1='\[\e[1;32m\]\u@\h\[\e[0m\]:\[\e[1;34m\]\w\[\e[0m\]\
$(git branch &>/dev/null; if [ $? -eq 0 ]; then \
  echo " \[\e[1;33m\]($(git branch | grep "^*" | cut -d" " -f2-))"; \
fi)\[\e[0m\]\$ '

# Right-aligned information
PS1='\[$(tput sc; printf "%*s" $COLUMNS "$(date +%H:%M)"; tput
↪  rc)\]\u@\h:\w\$ '
```

6 Signal Handling and Traps

6.1 Advanced Trap Techniques

Signal Handling

```
# Cleanup on script exit
trap 'rm -f "$TMPFILE"; exit' EXIT

# Ignore Ctrl-C
trap '' SIGINT

# Stackable traps
trap 'echo "First handler"' EXIT
trap 'echo "Second handler"' EXIT

# Process-specific traps
(
  trap 'echo "Child exiting"' EXIT
  sleep 5
)
```

6.2 Signal Reference

Table 21: Common UNIX Signals

Signal	Number	Purpose
SIGHUP	1	Hangup detection
SIGINT	2	Keyboard interrupt (Ctrl-C)
SIGQUIT	3	Quit with core dump
SIGKILL	9	Immediate termination
SIGTERM	15	Graceful termination
SIGTSTP	20	Terminal stop (Ctrl-Z)

7 Advanced Scripting Patterns

7.1 Singleton Pattern

Singleton Implementation

```
# Ensure single instance
LOCKFILE="/tmp/$(basename "$0").lock"
exec 9>"$LOCKFILE"
flock -n 9 || exit 1
```

7.2 Daemonization

Daemon Template

```
#!/bin/bash

# Daemon initialization
umask 0
cd /
setsid --fork >/dev/null 2>&1

# Main daemon loop
while true; do
    perform_task
    sleep $INTERVAL
done
```

Chapter Summary

This chapter explored advanced shell features including sophisticated I/O redirection, process substitution, parameter expansion, and array manipulation. We covered professional scripting patterns, signal handling, and shell customization techniques essential for system administrators.

Advanced Exercises

1. Create a script that uses coprocesses to maintain persistent database connections

2. Implement a recursive directory processor using globstar and null-terminated output
3. Develop a configuration file parser using associative arrays
4. Build a signal-aware daemon with proper cleanup handling
5. Design a custom prompt showing Git status, SSH connections, and load average

Chapter 5

User and Group Administration

Chapter Overview

This chapter provides a comprehensive guide to managing user and group accounts in UNIX systems. We'll cover account creation, password policies, privilege management, and security best practices for system administrators.

1 User Account Fundamentals

1.1 User Account Components

Table 22: User Account Configuration Files

File	Purpose
/etc/passwd	User account information
/etc/shadow	Secure password storage
/etc/group	Group definitions
/etc/skel/	Default user files
/etc/login.defs	Account creation defaults
/etc/default/useradd	User creation defaults

1.2 User Account Fields

/etc/passwd Entry Structure

```
username:x:UID:GID:GECOS:homedir:shell
```

Table 23: /etc/passwd Field Descriptions

Field	Example	Description
Username	jsmith	Login name
Password	x	Password placeholder
UID	1001	User ID number
GID	1001	Primary group ID
GECOS	John Smith	User information
Home	/home/jsmith	Home directory
Shell	/bin/bash	Login shell

2 User Account Management

2.1 Account Creation

User Creation Examples

```
# Basic user creation
useradd -m -c "John Smith" -s /bin/bash jsmith

# With specific UID/GID
useradd -u 1501 -g developers -G sudo,www-data jdoe

# Set password interactively
passwd jsmith

# Non-interactive password setting
echo "jsmith:password123" | chpasswd
```

2.2 Account Modification

User Modification Examples

```
# Change user properties
usermod -c "John Q. Smith" -l jqsmith jsmith

# Add to supplementary groups
usermod -aG sudo,adm jqsmith

# Lock/unlock account
usermod -L jqsmith  # Lock
usermod -U jqsmith  # Unlock

# Change home directory
usermod -d /new/home/jqsmith -m jqsmith
```

3 Password Policies

3.1 Password Aging Controls

Password Policy Examples

```
# Set password expiration
chage -M 90 -m 7 -W 14 jsmith

# Force password change on login
chage -d 0 jsmith

# View password aging
chage -l jsmith
```

Table 24: Password Policy Configuration

Option	Effect
-M days	Maximum password age
-m days	Minimum password age
-W days	Warning period
-I days	Inactive period
-E date	Expiration date

3.2 PAM Configuration

PAM Password Complexity

```
# /etc/pam.d/common-password
password requisite pam_pwquality.so retry=3 \
minlen=12 difok=3 ucredit=-1 lcredit=-1 dcredit=-1 ocredit=-1
```

4 Group Management

4.1 Group Operations

Group Management Examples

```
# Create new group
groupadd -g 2001 developers

# Modify group
groupmod -n devteam developers

# Delete group
groupdel devteam

# Add user to group
gpasswd -a jsmith devteam

# Remove user from group
gpasswd -d jsmith devteam
```

4.2 Group Membership Verification

Group Membership Checks

```
# List user groups
groups jsmith

# Check effective group membership
id jsmith

# Find all users in a group
getent group sudo | cut -d: -f4 | tr ',' '\n'
```

5 Privilege Management

5.1 sudo Configuration

/etc/sudoers Examples

```
# Allow user full sudo access
jsmith ALL=(ALL:ALL) ALL

# Allow group to run specific commands
%developers ALL=(ALL) /usr/bin/apt, /usr/bin/systemctl

# Passwordless sudo for specific command
%backup ALL=(ALL) NOPASSWD: /usr/bin/rsync

# Command aliases
Cmnd_Alias NETWORKING = /sbin/route, /sbin/ifconfig
%operators ALL=(ALL) NETWORKING
```

5.2 Best Practices for sudo

- Use `visudo` for editing sudoers file
- Grant minimal required privileges
- Prefer group-based over user-based rules
- Implement command restrictions where possible
- Log all sudo activity (`Defaults logfile="/var/log/sudo.log"`)

6 Account Security

6.1 Account Auditing

Security Audit Examples

```
# Find accounts with empty passwords
awk -F: '($2 == "") {print $1}' /etc/shadow

# Find non-root UID 0 accounts
awk -F: '($3 == 0) {print $1}' /etc/passwd | grep -v root

# Check last login times
lastlog | grep -v "Never logged in"

# Find inactive accounts
useradd -D | grep INACTIVE
find /home -type d -mtime +90 -exec ls -ld {} \;
```

6.2 Account Lockdown

Security Hardening

```
# Disable system accounts
usermod -s /sbin/nologin daemon
chage -E 0 daemon

# Set restrictive umask
echo "umask 027" >> /etc/profile

# Disable root SSH login
sed -i 's/^PermitRootLogin yes/PermitRootLogin no/'
  ↪ /etc/ssh/sshd_config

# Restrict cron access
echo "root" > /etc/cron.allow
rm -f /etc/cron.deny
```

7 Automated User Management

7.1 Bulk Operations

Bulk User Management

```
# Create users from list
while read -r user; do
  useradd -m "$user"
done < userlist.txt

# Reset passwords for all users
getent passwd | cut -d: -f1 | xargs -I{} chage -d 0 {}

# Disable expired accounts
awk -F: '$2 == "!!" {print $1}' /etc/shadow | xargs -I{} usermod -L
  ↪ {}
```

7.2 LDAP Integration

LDAP Configuration

```
# Install LDAP tools
apt install libnss-ldap libpam-ldap ldap-utils

# Configure nsswitch.conf
passwd:         files ldap
group:          files ldap
shadow:         files ldap

# Test LDAP lookup
getent passwd
ldapsearch -x -b "dc=example,dc=com"
```

Chapter Summary

This chapter covered comprehensive user and group administration techniques including account creation, password policies, privilege management, and security hardening. We explored both command-line tools and configuration files essential for system administrators.

Administration Exercises

1. Create a script that audits user accounts for password expiration and inactive sessions
2. Implement a sudo policy that allows developers to manage services but not modify system files
3. Configure PAM to enforce strong password policies (minimum length, complexity)
4. Design a bulk user import process from a CSV file
5. Harden system accounts by disabling shells and setting expiration dates

Chapter 6

File System Management

Chapter Overview

This chapter provides an in-depth examination of UNIX file system administration, covering disk partitioning, filesystem types, mounting strategies, performance optimization, and advanced management techniques essential for system administrators.

1 Disk Partitioning and Layout

1.1 Partition Table Types

Table 25: Partition Table Comparison

Type	Maximum Size	Features
MBR	2TB	Limited to 4 primary partitions
GPT	8ZB	Up to 128 partitions, CRC protection

1.2 Partitioning Tools

Partition Management Examples

```
# List block devices
lsblk -o NAME,SIZE,FSTYPE,MOUNTPOINT

# Create GPT partition table
parted /dev/sda mklabel gpt

# Create new partition
parted -a optimal /dev/sda mkpart primary ext4 0% 100%

# Verify partition alignment
parted /dev/sda align-check optimal 1
```

2 File System Types and Features

2.1 Common UNIX File Systems

Table 26: File System Comparison

File System	Strengths	Ideal Use Case
ext4	Stable, journaling	General purpose
XFS	High performance, scalability	Large files, media
Btrfs	Snapshots, checksums	Data integrity
ZFS	Advanced features, compression	Enterprise storage
tmpfs	Memory-backed	Temporary files

2.2 File System Creation

File System Operations

```
# Create ext4 filesystem
mkfs.ext4 -L datavolume /dev/sda1

# Create XFS filesystem
mkfs.xfs -f -L bigdata /dev/sdb1

# Add ZFS storage pool
zpool create datapool mirror /dev/sdc /dev/sdd

# Check filesystem integrity
fsck -y /dev/sda1
```

3 Mounting File Systems

3.1 Mount Options and Strategies

Table 27: Common Mount Options

Option	Purpose
noatime	Disable access time updates
nodiratime	Disable directory access time
relatime	Optimized access time updates
discard	Enable TRIM (SSDs)
data=journal	Full data journaling
barrier=1	Write barrier enforcement

Mount Examples

```
# Temporary mount
mount /dev/sdb1 /mnt/data

# Persistent mount (add to /etc/fstab)
UUID=1234-5678 /data ext4 defaults,noatime 0 2

# Bind mount
mount --bind /var/www /srv/www

# Remount with new options
mount -o remount,ro /dev/sda1
```

4 Advanced File System Features

4.1 Logical Volume Management

LVM Operations

```
# Initialize physical volume
pvcreate /dev/sdb

# Create volume group
vgcreate vg_data /dev/sdb

# Create logical volume
lvcreate -L 100G -n lv_www vg_data

# Extend logical volume
lvextend -L +50G /dev/vg_data/lv_www
resize2fs /dev/vg_data/lv_www
```

4.2 Quota Management

Disk Quota Setup

```
# Enable quotas in fstab
UUID=1234-5678 /home ext4 defaults,usrquota,grpquota 0 2

# Initialize quota files
quotacheck -cug /home
quotaon /home

# Set user quotas
setquota -u jsmith 500M 1G 0 0 /home

# Generate quota reports
repquota -a
```

5 File System Maintenance

5.1 Monitoring and Analysis

Monitoring Examples

```
# Check disk space
df -hT -x tmpfs

# Find large files
find / -xdev -type f -size +100M -exec ls -lh {} \+

# Analyze disk usage
ncdu -x /

# Check inode usage
df -i
```

5.2 Scheduled Maintenance

Maintenance Script

```
#!/bin/bash
# Weekly filesystem maintenance
logger -t maint "Starting filesystem maintenance"

# Rotate logs
logrotate -f /etc/logrotate.conf

# Trim SSD
fstrim -av

# Check filesystems
fsck -A -C -t ext4 -p

# Update locate database
updatedb

logger -t maint "Completed filesystem maintenance"
```

6 Backup and Recovery

6.1 Backup Strategies

Table 28: Backup Method Comparison

Method	Advantages	Limitations
Full	Complete recovery	Storage intensive
Incremental	Efficient storage	Complex restoration
Differential	Balanced approach	Growing backup size
Snapshot	Instant recovery	Requires COW filesystem

6.2 Backup Tools

Backup Examples

```
# Full backup with tar
tar -czpf /backups/full-$(date +\%F).tar.gz --exclude=/backups /

# Incremental backup with rsync
rsync -avh --delete --link-dest=/backups/last_full /
  ↪ /backups/incr-$(date +\%F)

# Filesystem snapshot
lvcreate -s -n db_snap -L 10G /dev/vg_data/lv_database

# Database dump
mysqldump -u root -p --all-databases | gzip > /backups/mysql-$(date
  ↪ +\%F).sql.gz
```

7 Security and Permissions

7.1 Advanced Permission Management

Permission Examples

```
# Set default permissions
setfacl -d -m u::rwx,g::r-x,o::r-x /shared

# Recursive permission fix
find /webroot -type d -exec chmod 755 {} \;
find /webroot -type f -exec chmod 644 {} \;

# Special permissions
chmod +t /tmp/uploads  # Sticky bit
chmod g+s /var/www     # Setgid
```

7.2 File Attributes

File Attributes

```
# Make file immutable
chattr +i /etc/passwd

# Secure deletion attribute
chattr +s important.doc

# Append-only logs
chattr +a /var/log/secure

# View attributes
lsattr /etc/ssh/sshd_config
```

Chapter Summary

This chapter covered comprehensive file system management including partitioning strategies, filesystem types, mounting options, advanced features like LVM and quotas, maintenance procedures, and backup solutions. These techniques form the foundation of reliable storage administration.

Administration Exercises

1. Design a partitioning scheme for a database server with separate partitions for OS, logs, and data
2. Implement a monitoring system that alerts when filesystem usage exceeds 90%
3. Create an automated backup rotation script with full and incremental backups
4. Configure a secure shared directory with proper permissions and quotas
5. Test recovery procedures by restoring from backup to a test system

Chapter 7

Process and Service Management

Chapter Overview

This chapter provides a comprehensive guide to UNIX process and service management, covering process monitoring, control, prioritization, systemd service units, and automation techniques essential for system administrators.

1 Process Fundamentals

1.1 Process States

Table 29: UNIX Process States

State	Description
R (Running)	Currently executing or runnable
S (Sleeping)	Waiting for an event
D (Uninterruptible)	Waiting on I/O (cannot be killed)
Z (Zombie)	Terminated but not reaped
T (Stopped)	Suspended by signal

1.2 Process Hierarchy

Process Tree Examples

```
# View process hierarchy
pstree -p

# Show parent-child relationships
ps -ef --forest

# Find parent process ID
ps -o ppid= -p [PID]
```

2 Process Monitoring

2.1 Monitoring Tools

Table 30: Process Monitoring Utilities

Command	Purpose
ps	Snapshot of processes
top	Interactive process viewer
htop	Enhanced interactive viewer
vmstat	System resource statistics
pidstat	Per-process statistics

2.2 Advanced Monitoring

Monitoring Examples

```
# Show threads
ps -eLf

# Monitor process memory
pmap -x [PID]

# Continuous I/O monitoring
iotop -oPa

# Process-specific CPU usage
pidstat -u -p [PID] 5 3
```

3 Process Control

3.1 Signals and Termination

Table 31: Common Process Signals

Signal	Number	Purpose
SIGTERM	15	Graceful termination
SIGKILL	9	Forceful termination
SIGSTOP	19	Suspend process
SIGCONT	18	Resume process
SIGHUP	1	Reload configuration

Process Control Examples

```
# Graceful shutdown
kill -TERM [PID]

# Force kill
kill -9 [PID]

# Kill by name
pkill -f "pattern"

# Kill all matching processes
killall -9 process_name
```

4 Process Prioritization

4.1 Nice and Renice

Priority Management

```
# Start process with low priority
nice -n 19 cpu_intensive_task

# Change running process priority
renice -n 10 -p [PID]

# Show nice values
ps -eo pid,ni,comm
```

4.2 Scheduling Classes

Table 32: Process Scheduling Policies

Policy	Description
SCHED_OTHER	Default time-sharing
SCHED_FIFO	Real-time (first-in, first-out)
SCHED_RR	Real-time (round robin)
SCHED_BATCH	For batch processes
SCHED_IDLE	For very low priority tasks

Scheduling Examples

```
# Set FIFO scheduling
chrt -f -p 99 [PID]

# Run task with batch scheduling
chrt -b 0 batch_task
```

5 Systemd Service Management

5.1 Service Unit Files

Service Unit Example

```
# /etc/systemd/system/custom.service
[Unit]
Description=Custom Service
After=network.target

[Service]
Type=simple
User=svcuser
Group=svcgroup
ExecStart=/usr/local/bin/service_start
ExecStop=/usr/local/bin/service_stop
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

5.2 Service Commands

Systemd Examples

```
# Start/stop service
systemctl start servicename
systemctl stop servicename

# Enable/disable at boot
systemctl enable servicename
systemctl disable servicename

# Check status
systemctl status servicename

# Reload modified units
systemctl daemon-reload
```

6 Logging and Debugging

6.1 Journalctl Usage

Journal Examples

```
# Show service logs
journalctl -u servicename

# Follow logs in real-time
journalctl -f

# Filter by priority
journalctl -p err

# Show kernel messages
journalctl -k

# Persistent logging
mkdir /var/log/journal
systemd-tmpfiles --create --prefix /var/log/journal
```

6.2 Process Tracing

Debugging Examples

```
# Trace system calls
strace -p [PID]

# Trace child processes
strace -f command

# Monitor file access
lsof -p [PID]

# Network connections
ss -tupn
```

7 Automation and Scheduling

7.1 Cron Jobs

Cron Examples

```
# Edit crontab
crontab -e

# Example entries
0 2 * * * /path/to/backup.sh
*/5 * * * * /path/to/monitor.sh

# System-wide cron
vim /etc/crontab
```

7.2 Systemd Timers

Timer Unit Example

```
# /etc/systemd/system/daily-backup.timer
[Unit]
Description=Daily Backup Timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target

# /etc/systemd/system/daily-backup.service
[Unit]
Description=Daily Backup

[Service]
Type=oneshot
ExecStart=/path/to/backup.sh
```

8 Resource Limits

8.1 ulimit Configuration

Resource Limits

```
# View limits
ulimit -a

# Set per-session limits
ulimit -n 4096  # Open files
ulimit -u 500   # User processes

# System-wide limits
vim /etc/security/limits.conf
* soft nofile 4096
* hard nofile 8192
```

8.2 Cgroups v2

Cgroup Examples

```
# Create new cgroup
mkdir /sys/fs/cgroup/mycgroup

# Set memory limit
echo "100000000" > /sys/fs/cgroup/mycgroup/memory.max

# Add process to cgroup
echo [PID] > /sys/fs/cgroup/mycgroup/cgroup.procs
```

Chapter Summary

This chapter covered comprehensive process and service management including monitoring, control, prioritization, systemd services, logging, automation, and resource limits. These skills are essential for maintaining optimal system performance and reliability.

Administration Exercises

1. Create a systemd service unit for a custom application with auto-restart
2. Write a script to identify and kill zombie processes
3. Configure a CPU-intensive process to use idle scheduling
4. Implement log rotation for a service using journald
5. Set up a systemd timer for weekly maintenance tasks